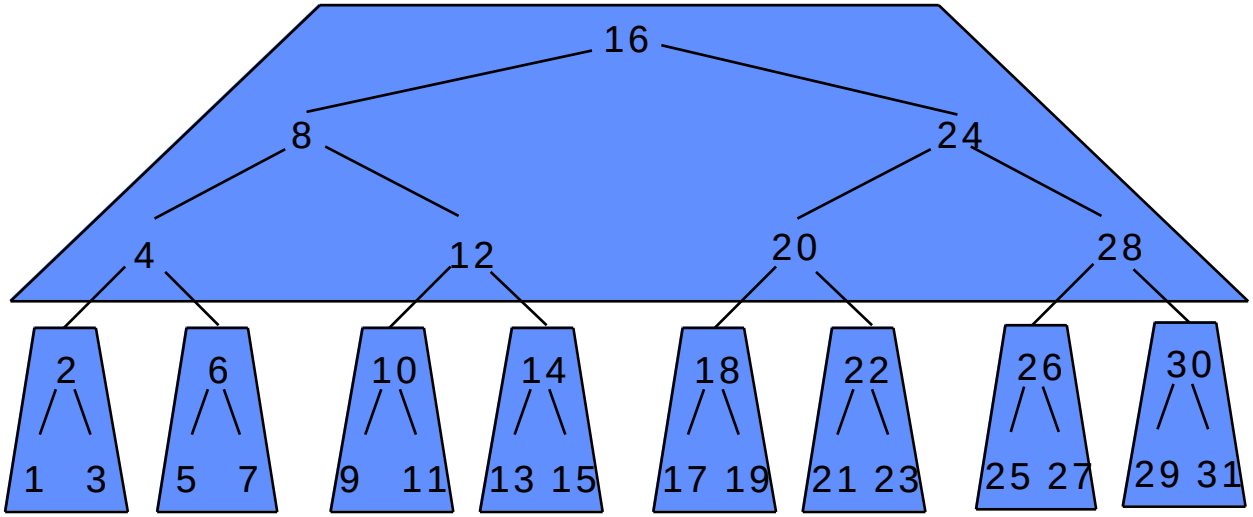
Arbres balancés

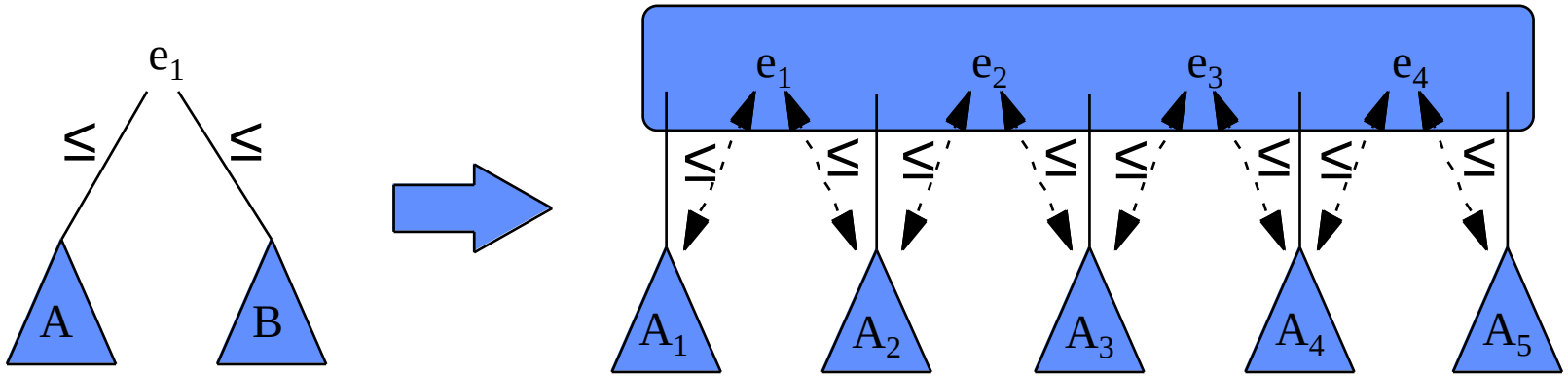
Le problème

- prendre en compte la taille des blocs disques
  - peuvent contenir plusieurs nœuds
  - temps d'accès environ 20 ms
- regrouper les nœuds voisins dans un même bloc
  - difficile à maintenir après adjonction ou suppression



La solution

- mettre plusieurs valeurs par nœuds
  => arbres généraux de recherche
  - ♦ les nœuds sont des listes (ordonnées) de valeurs
  - ♦ chaque valeur est un séparateur entre deux sous-arbres
  - ♦ apporter des contraintes sur la construction des arbres généraux



arbre binaire de recherche *arbre général de recherche*

Aperçu de la spécification

type arbregen-rech **dérive de** arbregen, forêt-rech **dérive de** forêt

opérations

max : arbregen-rech → clé

min : arbregen-rech → clé

ordonné : nœud × forêt-gen → booléen

préconditions

<o, f >: ordonné (o, f)

axiomes

max(<o, f>) = **si** f = Ø **alors** la_clé(ième(o, longueur(o)))
sinon max(ième(f, longueur(f))) **fsi**

min(<o, f>) = **si** f = Ø **alors** la_clé(ième(o, 1)) **sinon** min(ième(f, 1)) **fsi**

ordonné(o, f) = **si** estvide (o) **alors** faux
sinsi f = Ø **alors** est-triée(o)
sinsi longueur(f) ≠ longueur(o) + 1 **alors** faux
sinsi o = [e] f = [a;b] **alors** max(a) ≤ la_clé(e) ≤ min(b)
sinon soit o = e::o', f = a::f';
max(a) ≤ la_clé(e) ≤ min(ième(f',1)) ordonné(o',f') **fsi**

Implantation en Ada


```
with Listes_Contigues_Ordonnees; with Listes_Contigues;
generic Degre_Max : in Positive := 100;
    type Id_Noeud is private; type Element is private; type Cle is private;
    with function LaCle (E : in Element) return Cle is <>;
    with function "<"(U, V: in Cle) return Boolean is <>;
package Arbres_Balances is
    type Type_Noeud is private;
private
    package L_E is new Listes_Contigues_Ordonnees (Element, Cle);
    subtype Liste_Valeurs is L_E.Liste (Degre_Max);
    package L_P is new Listes_Contigues (Id_Noeud);
    subtype Liste_Fils is L_P.Liste (Degre_Max + 1);
    type Type_Noeud is record Contenu : Liste_Valeurs;
                              Sous_Arbres : Liste_Fils;
                       end record;
end Arbres_Balances;
```

La recherche

extension type arbregen-rech

opérations **préconditions**

rang : clé × arbregen-rech → entier

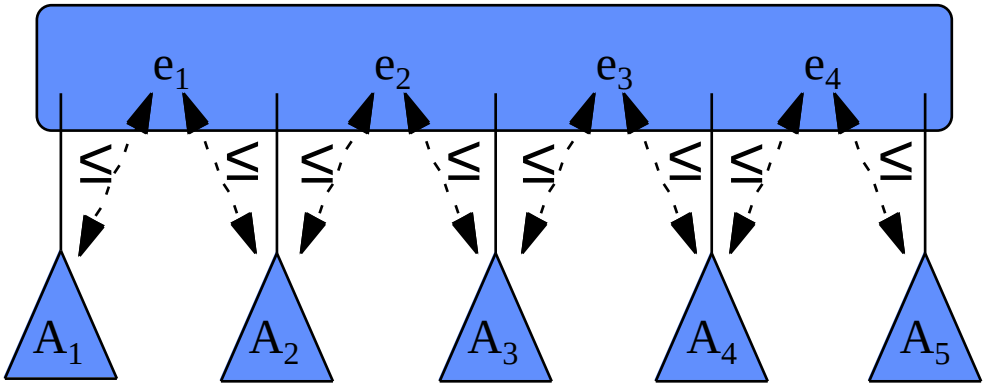
recherche : clé × arbregen-rech /→ élément c a

axiomes

rang(c, a) = recherche (c, racine (a)) ← *recherche dans une liste ordonnée*

recherche (c, a) = **soit** a = <o, f>, i = rang (c, a);
si i ≤ d°(a) c = la_clé(ième(o, i)) **alors** ième(o, i)
sinon recherche (c, ième (f, i)) **fsi**

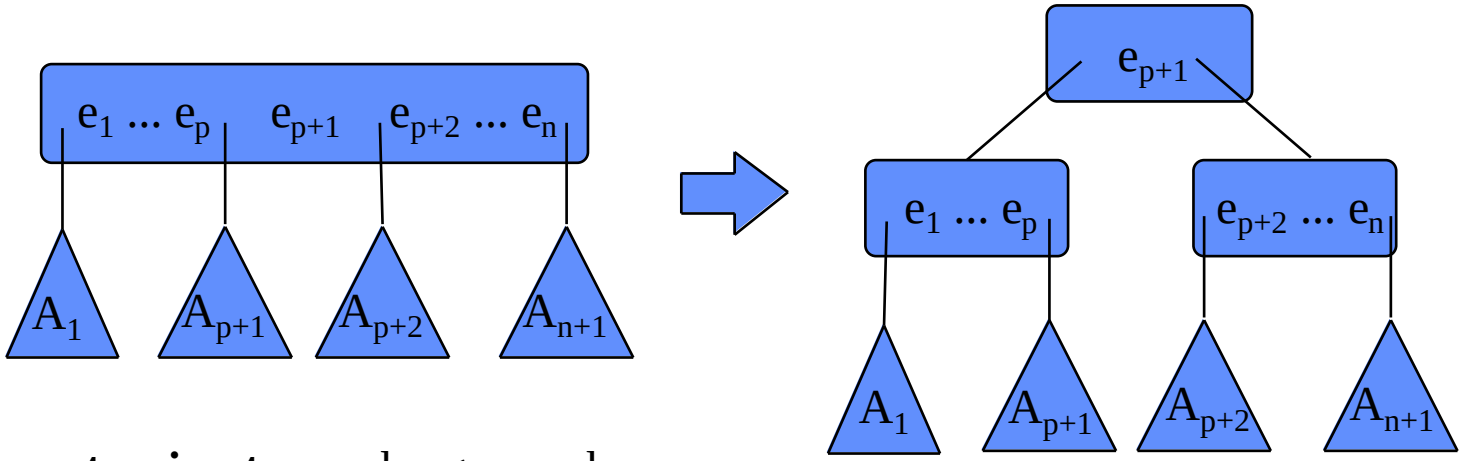
- complexité: h(a) si rang est une opération fondamentale



Opérations de rééquilibrage

- But: maintenir le nombre de valeurs par nœud dans des limites raisonnables
- Comment?
  - éclatements
  - regroupements
  - répartitions

Eclatement d'une racine



extension type arbregen-rech

opérations **préconditions**

éclater : arbregen-rech / → arbregen-rech d°(a) ≥ 3

axiomes

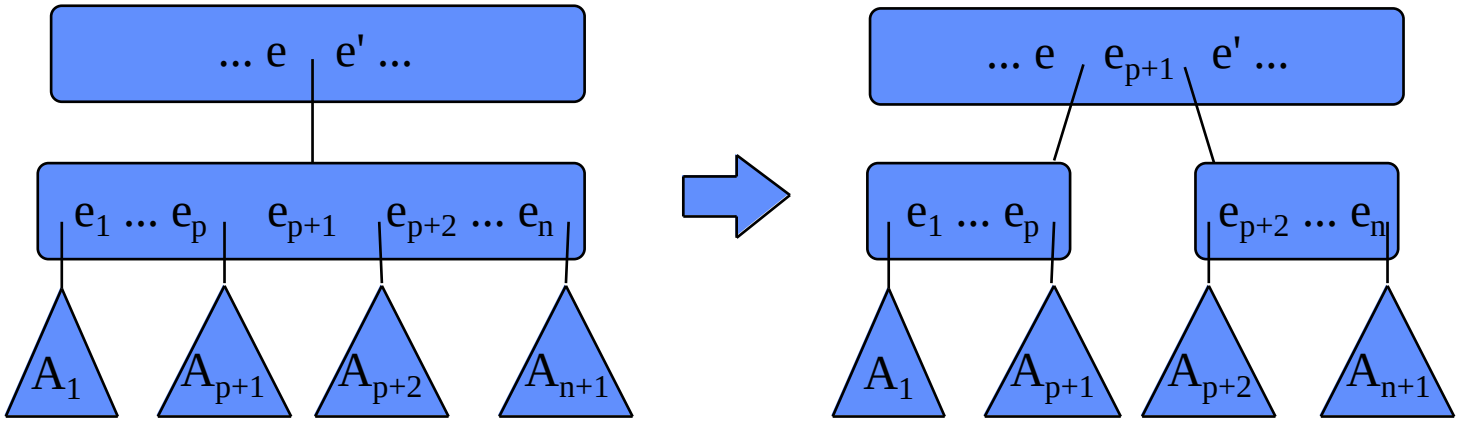
eclater (a) = <[e], [u;v]> **tels que**

½d°(u)-d°(v)½≤1

racine(a) = racine (u) & [e] & racine (v)

sousarbres(a) = sousarbres (u) & sousarbres (v)

Eclatement d'un sous-arbre



extension type arbregen-rech

opérations

éclater-ième : arbregen-rech × entier /→ arbregen-rech

préconditions

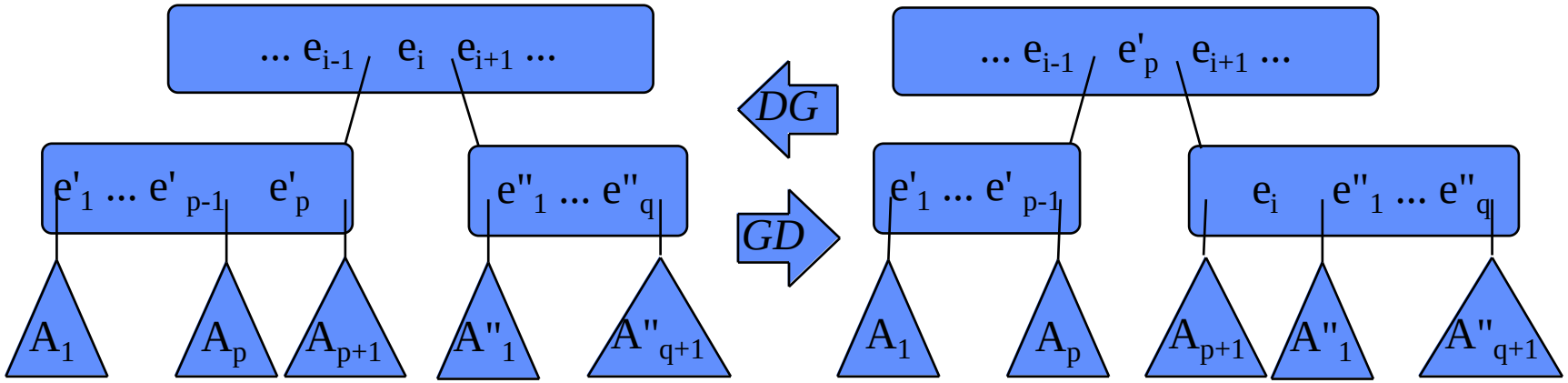
éclater-ième(a, i): 1 ≤ i ≤ d°(a)+1 sousarbres(a) ≠ Ø d°(ième(sousarbres(a), i)) ≥ 3

axiomes

éclater-ième(<o, f>, i) = **soit** <[e], [u;v]> = éclater(ième(f, i));
<insérer(o, i, e), insérer(changer-ième(f, i, u), i+1, v)>

1 éclatement = 3 écritures *regroupement = opération inverse*

Transferts et répartition



- transfert gauche-droite et transfert droite-gauche
  - noter l'analogie avec les rotations dans les arbres AVL
  - on peut répéter les transferts pour rapprocher les degrés des sous-arbres
- répartition = regroupement puis éclatement
  - les deux sous-arbres sont de même degré à 1 près

Arbres balancés ou B-arbres

- Un B-arbre d'ordre m est:
  - un arbre général de recherche
  - toutes les feuilles sont au même niveau
  - tout nœud contient au plus 2m valeurs
  - tout nœud sauf la racine contient au moins m valeurs

$$n \leq 2m\,(1 + (2m+1) + (2m+1)^2 + \ldots + (2m+1)^h) = (2m+1)^{h+1} - 1$$

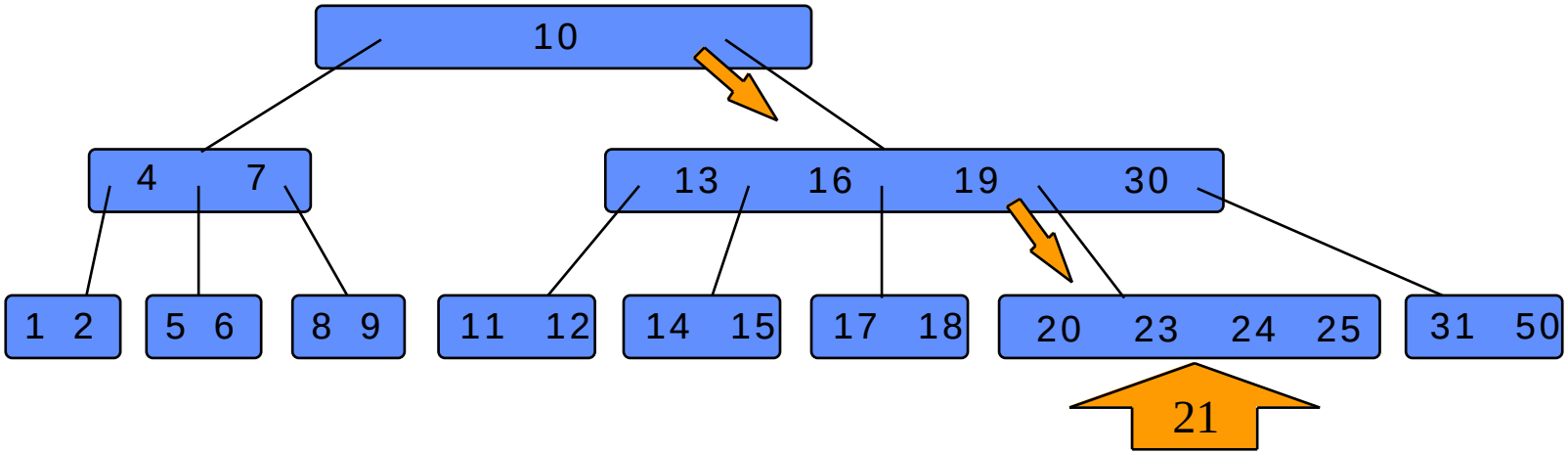
$$n \geq 1 + 2m\,(1 + (m+1) + (m+1)^2 + \ldots + (m+1)^{h-1}) = 2(m+1)^h - 1$$

$$\Rightarrow \log_{2m+1}(n+1) \leq h \leq \log_{m+1}(n+1)/2$$

- Exemples:
  - m = 100 et h = 2 => 20 000 ≤ n ≤ 8 000 000
  - m = 100 et h = 3 => 2 000 000 ≤ n ≤ 1 600 000 000

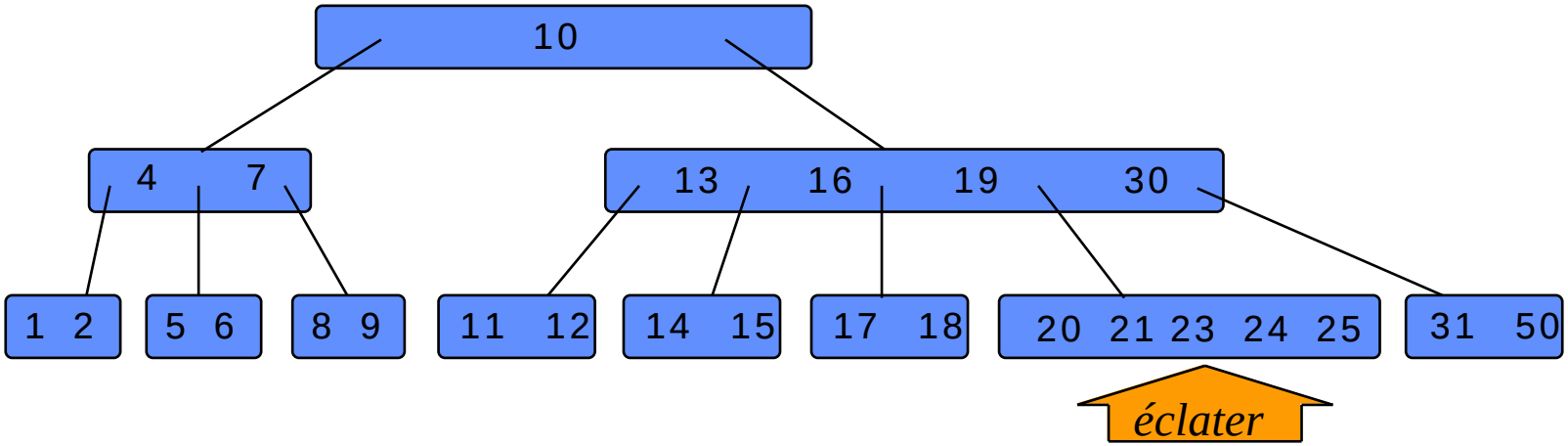
Adjonction (1)

- Ajouter 21 à l'arbre suivant
- Recherche du nœud



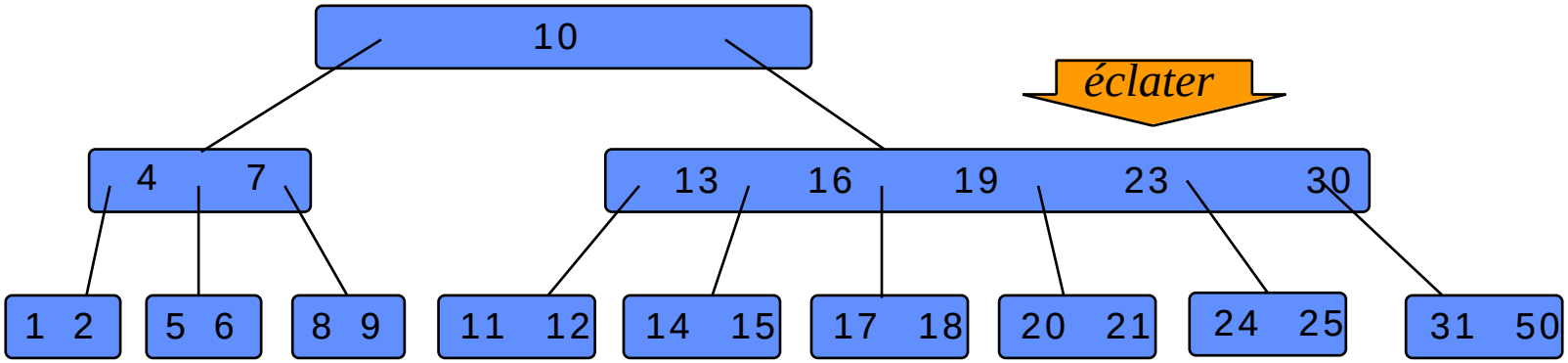
Adjonction (2)

- Le nœud contient 5 valeurs, ce qui est trop => éclater
- Note: on pourrait répartir



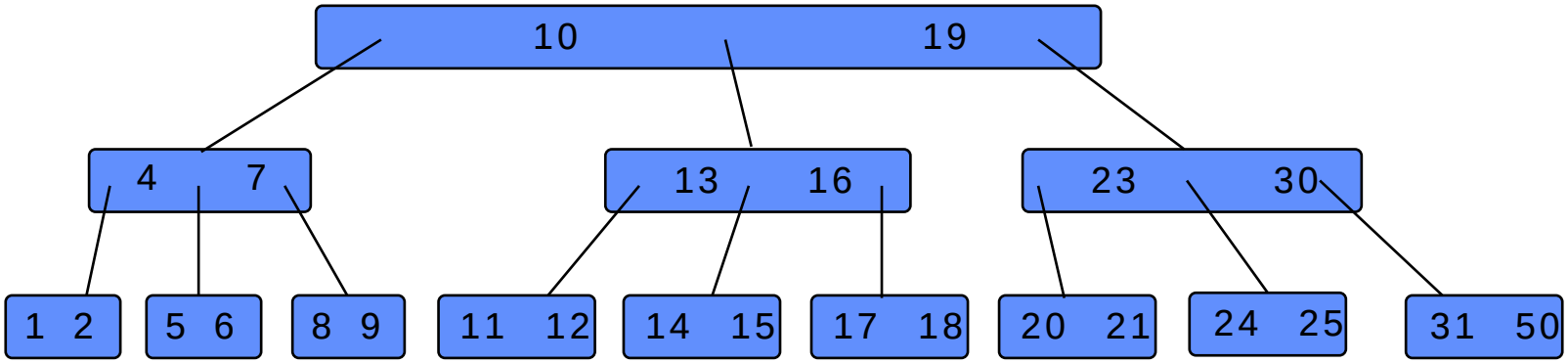
Adjonction (3)

- 23 monte dans le nœud père qui contient donc 5 valeurs, ce qui est trop => il faut l'éclater
- Note: ici encore, on pourrait répartir



Adjonction (4)

- 19 monte à la racine, qui contient 2 éléments => correct
- Complexité: hauteur de l'arbre
- Pile des ancêtres pour les éclatements éventuels



Adjonction (5)

extension type B-arbre
opérations **préconditions**
ajouter : élément × B-arbre /→ B-arbre ¬ (la_clé(e) a)
ajouter-int : élément × B-arbre → arbregen-rech
axiomes
ajout-int (e, a) =
soit a = <o, f>, i = rang (la_cle (e), a);
si f = Ø **alors** <insérer(o, i, e), Ø>
sinon soit u = ajout-int(e, ième(f, i)); f'= changer-ième(f, i, u);
si d°(u) ≤ 2m **alors** <o, f'> **sinon** éclater-ième(<o, f'>, i) **fsi**
fsi
ajouter (e, a) = **soit** u = ajout-int (e, a);
si d°(u) ≤ 2m **alors** u **sinon** eclater (u) **fsi**

adjonction dans le sous-arbre i

puis éclatement si nécessaire

adjonction proprement dite, puis correction éventuelle → *séparer en deux opérations*

Adjonction (6)

extension type B-arbre

opérations **préconditions**

corriger-a : arbregen-rech × entier /→ arbregen-rech 1≤i≤d°(a)+1 a≠<o,Ø>

ajouter : élément × B-arbre /→ B-arbre ¬ (la_clé(e) a)

ajouter-int : élément × B-arbre → arbregen-rech

axiomes

corriger-a(a, i) = **soit** a = <o, f>; *opération de correction après adjonction*

si d°(ième(f, i) ≤ 2m **alors** a

sinsi i > 1 d°(ième(f, i-1) < 2m **alors** transfert-droite-gauche (a, i-1)

sinsi i ≤ d°(a) d°(ième(f, i+1) < 2m **alors** transfert-gauche-droite (a, i)

sinon éclater-ième (a, i) **fsi**

ajout-int (e, a) = **soit** a = <o, f>, i = rang (la_cle (e), a);

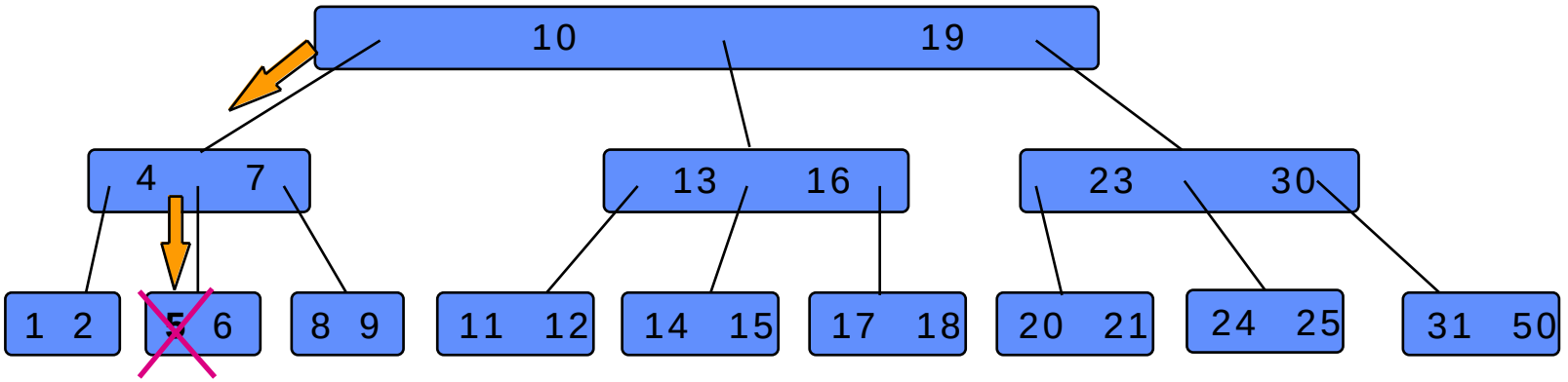
si f = Ø **alors** <insérer(o, i, e), Ø>

sinon soit f'= changer-ième(f,i,ajout-int(e,ième(f,i))); corriger-a(<o, f'>, i) **fsi**

ajouter (e, a) = **soit** u = ajout-int (e, a); **si** d°(u) ≤ 2m **alors** u **sinon** eclater (u) **fsi**

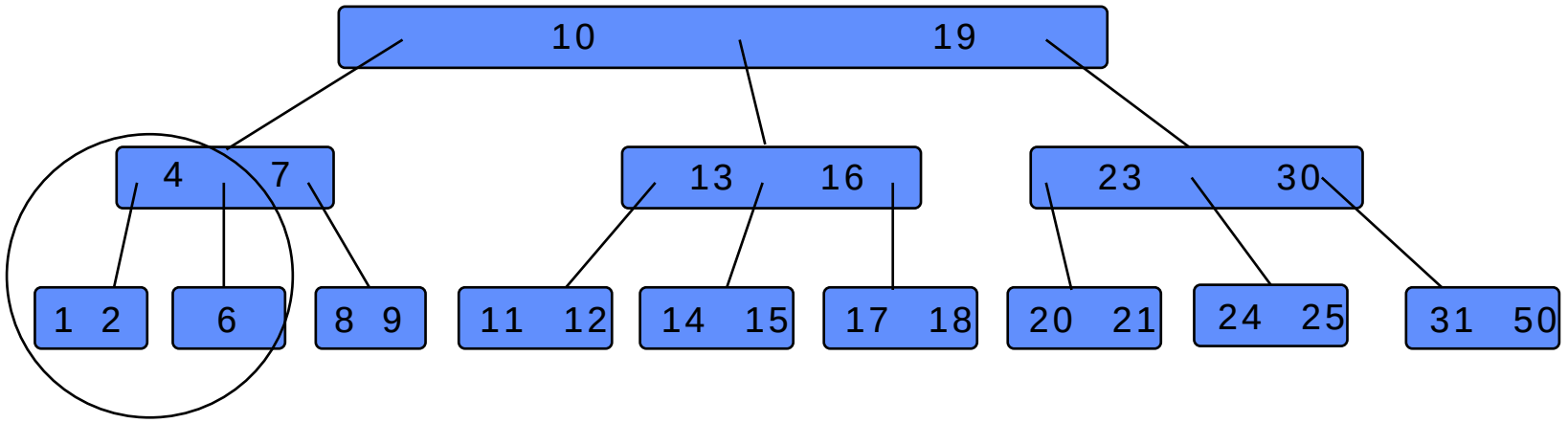
Suppression (1)

- suppression dans un nœud non feuille => remplacer la valeur par la plus petite du (i+1)ème sous-arbre
- suppression effective dans une feuille
- Exemple: suppression de 5
  - recherche de 5



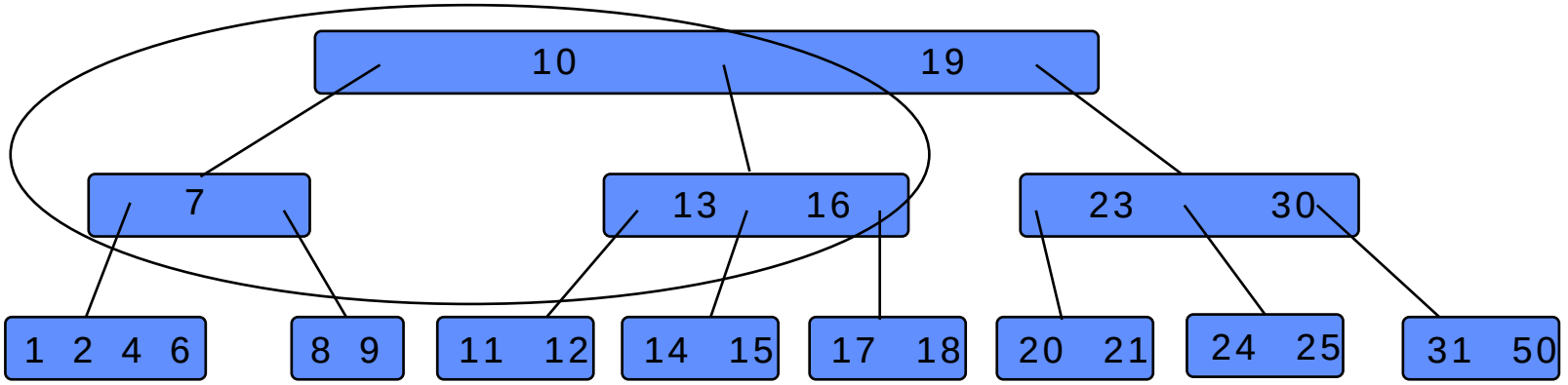
Suppression (2)

- Le nœud ne contient plus qu'une valeur, donc trop petit
- Les deux nœuds voisins sont au minimum, => regroupement



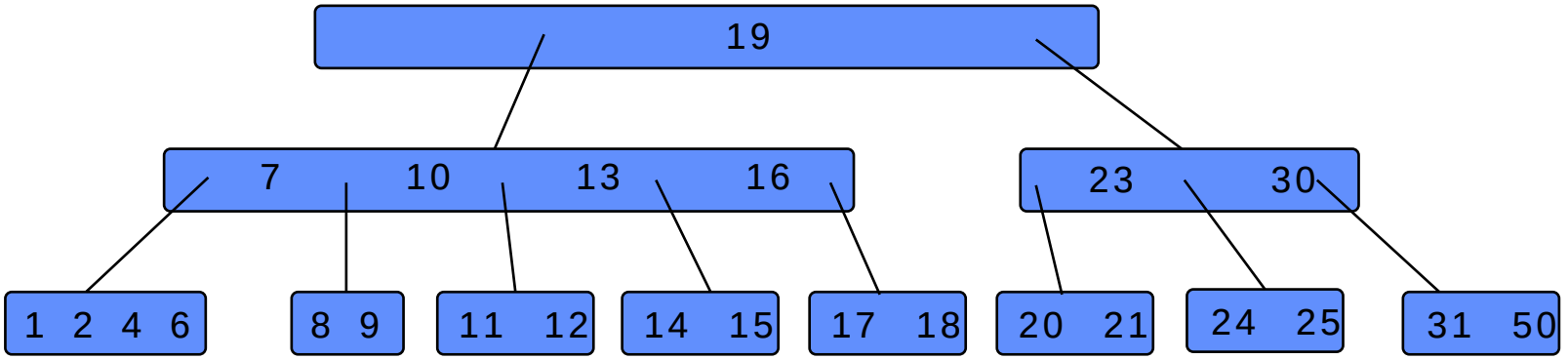
Suppression (3)

- 4 descend, laissant une valeur dans le nœud, donc trop petit
- Son unique voisin est au minimum, donc regroupement



Suppression (4)

- 10 descend, et laisse une seule valeur dans le nœud, mais il s'agit de la racine
- complexité: hauteur de l'arbre
- nécessité de conserver le chemin depuis la racine



Suppression (5)

extension type B-arbre

opérations		**préconditions**
corriger-s	: arbregen-rech × entier /→ arbregen-rech	1≤i≤d°(a)+1 a≠<o,Ø>
gauche	: B-arbre → nœud	
sup-gauche	: B-arbre → arbregen-rech	
supprimer	: clé × B-arbre /→ B-arbre	c a

axiomes

corriger-s (a, i) = **soit** a = <o, f>;
si d°(ième(f, i) ≥ m **alors** a
sinsi i ≤ d°(a) d°(ième(f, i+1) > m **alors** transfert-droite-gauche (a, i)
sinsi i > 1 d°(ième(f, i-1) > m **alors** transfert-gauche-droite (a, i-1)
sinsi i > 1 **alors** regrouper (a, i-1)
sinon regrouper (a, i)
fsi

Suppression (6)

axiomes

gauche (<o, f>) = **si** f = Ø **alors** ième (o, 1) **sinon** gauche (ième (f, 1)) **fsi**

sup-gauche (<o, f>) = **si** f = Ø **alors** <supprimer (o, 1), Ø>

sinon soit f' = changer-ième(f,1,sup-gauche(ième(f, 1))); corriger-s (<o, f'>, 1) **fsi**

supprimer (c, <o, f>) = **soit** i = rang(c, <o, f>);

si f = Ø **alors** <supprimer (o, i), Ø>

sinsi i ≤ d°(<o, f>) c = la_cle(ième(o, i)) **alors**

soit o' = changer-ième(o, i, gauche(ième(f, i+1))),

f' = changer-ième(f, i+1, sup-gauche(ième(f, i+1)));

corriger-s (<o', f'>, i+1)

sinon soit f' = changer-ième(f, i, supprimer(c, ième(f, i)));

corriger-s (<o, f'>, i)

fsi

Conclusion

- maintient de la propriété B-arbre:
  - adjonction => éclatement ou répartition
  - suppression => regroupement ou répartition
- rang est une opération bornée: $\Theta(m)$ ou $\Theta(\log m)$
- toutes les opérations: au plus $\log_{m+1}(n+1)/2$ accès disque
- en moyenne 1 éclatement pour 1.38m adjonctions
- taux de remplissage moyen: 0.7
- amélioration: B-arbre sur les clés, pour augmenter m
- applications: fichiers séquentiels indexés

Implantation

- parcours en liste infixe
  - chemin des blocs depuis la racine
  - position de la valeur dans le bloc
  - position du bloc sur disque
- adjonctions/suppressions
  - disposer du bloc gauche ou droit
- éviter la relecture d'un bloc en mémoire